The Im2Col Transformation:
A Speedrunner's Guide to CNNs

Francesco Pio Crispino   Massimiliano Romani

Artificial Intelligence and Data Engineering
University of Pisa

Academic Year 2024/2025

Contents

Introduction

What is a CNN?

A Convolutional Neural Network is a NN designed to find patterns in everything, but it's mostly used to extract features from images.

As the name suggests, this process is achieved with the convolution operation

Convolution in theory

Discrete Convolution

Given **f** the input image and **g** the filter (or kernel), the $[i, j]$-th element of the matrix **C**, result of the convolution between f and g, is computed as:

$$C[i, j] = (f \star g)[i, j] = \sum_{m} \sum_{n} f[i + m, j + n] \cdot g[m, n]$$

Convolution in practice

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \circledast \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} \xrightarrow{s=1,p=0} \begin{bmatrix} \begin{array}{c} 1 \cdot 10 + 2 \cdot 11 + \\ 4 \cdot 12 + 5 \cdot 13 \end{array} & \begin{array}{c} 2 \cdot 10 + 3 \cdot 11 + \\ 5 \cdot 12 + 6 \cdot 13 \end{array} \\ \\ \begin{array}{c} 4 \cdot 10 + 5 \cdot 11 + \\ 7 \cdot 12 + 8 \cdot 13 \end{array} & \begin{array}{c} 5 \cdot 10 + 6 \cdot 11 \\ 8 \cdot 12 + 9 \cdot 13 \end{array} \end{bmatrix} = \begin{bmatrix} 145 & 191 \\ 283 & 329 \end{bmatrix}$$

Padding operation

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow[\text{padding=1}]{\text{zero padding}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Stride

- **input**: 4x4 matrix
- stride=1 with 2x2 kernel
- **result**: 3x3 matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \dots$$

Stride

- **input**: 4x4 matrix
- stride=2 with 2x2 kernel
- **result**: 2x2 matrix

$$\begin{bmatrix} \color{red}{1} & \color{red}{2} & 3 & 4 \\ \color{red}{5} & \color{red}{6} & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & \color{red}{3} & \color{red}{4} \\ 5 & 6 & \color{red}{7} & \color{red}{8} \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \color{red}{9} & \color{red}{10} & 11 & 12 \\ \color{red}{13} & \color{red}{14} & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \color{red}{11} & \color{red}{12} \\ 13 & 14 & \color{red}{15} & \color{red}{16} \end{bmatrix}$$

Output Dimensions

The Output dimensions of a convolution are computed as follows:

- **Output Height** $= \lfloor \frac{(Image_{Height} - Kernel_{Height} + 2 \cdot Padding)}{Stride} \rfloor + 1$
- **Output Width** $= \lfloor \frac{(Image_{Width} - Kernel_{Width} + 2 \cdot Padding)}{Stride} \rfloor + 1$

Dilation operation

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{\text{dilate} = 1} \begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 \end{bmatrix}$$

im2col

The trick

im2col transforms a convolution into a matrix multiplication

$$\begin{bmatrix} \color{red}{1} & \color{red}{2} & 3 \\ \color{red}{4} & \color{red}{5} & 6 \\ 7 & 8 & 9 \end{bmatrix} \circledast \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} \quad = \quad \begin{bmatrix} \color{red}{1} & \color{red}{2} & \color{red}{4} & \color{red}{5} \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \end{bmatrix} \longrightarrow \begin{bmatrix} 145 \\ 191 \\ 283 \\ 329 \end{bmatrix} \longrightarrow \begin{bmatrix} 145 & 191 \\ 283 & 329 \end{bmatrix}$$

Multiple Channels

- First Channel
- Second Channel
- N. of image channels MUST always match N.of each kernel channels

$$\begin{matrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \end{matrix} \circledast \begin{matrix} \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} \\ \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} \end{matrix} = \begin{bmatrix} 1 & 2 & 4 & 5 & 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 & 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 & 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 & 5 & 6 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 10 \\ 11 \\ 12 \\ 13 \end{bmatrix} = \begin{bmatrix} 290 \\ 382 \\ 566 \\ 658 \end{bmatrix} = \begin{bmatrix} 290 & 382 \\ 566 & 658 \end{bmatrix}$$

Multiple Filters

- Each convolution with a kernel generates a channel for the output image

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \circledast \begin{bmatrix} {\color{green}10} & {\color{green}11} \\ {\color{green}12} & {\color{green}13} \end{bmatrix}, \begin{bmatrix} {\color{red}14} & {\color{red}15} \\ {\color{red}16} & {\color{red}17} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} {\color{green}10} & {\color{red}14} \\ {\color{green}11} & {\color{red}15} \\ {\color{green}12} & {\color{red}16} \\ {\color{green}13} & {\color{red}17} \end{bmatrix} = \begin{bmatrix} {\color{green}145} & {\color{red}193} \\ {\color{green}191} & {\color{red}255} \\ {\color{green}283} & {\color{red}379} \\ {\color{green}329} & {\color{red}441} \end{bmatrix} = \begin{matrix} \begin{bmatrix} {\color{green}145} & {\color{green}191} \\ {\color{green}283} & {\color{green}329} \end{bmatrix} \\ \begin{bmatrix} {\color{red}193} & {\color{red}255} \\ {\color{red}379} & {\color{red}441} \end{bmatrix} \end{matrix}$$

Multiple Images

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}, \begin{bmatrix}10 & 11 & 12\\13 & 14 & 15\\16 & 17 & 18\end{bmatrix} \circledast \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} = \begin{bmatrix}1 & 2 & 4 & 5\\2 & 3 & 5 & 6\\4 & 5 & 7 & 8\\5 & 6 & 8 & 9\\10 & 11 & 13 & 14\\11 & 12 & 14 & 15\\13 & 14 & 16 & 17\\14 & 15 & 17 & 18\end{bmatrix} \times \begin{bmatrix}1\\2\\3\\4\end{bmatrix} = \begin{bmatrix}37\\47\\67\\77\\127\\137\\157\\167\end{bmatrix} = \begin{bmatrix}37 & 47\\67 & 77\end{bmatrix}, \begin{bmatrix}127 & 137\\157 & 167\end{bmatrix}$$

The Problem

Handwritten Digit Classification

To test the approach, the task of classifying MNIST's handwritten digit was chosen, because of its simplicity and widely known dataset

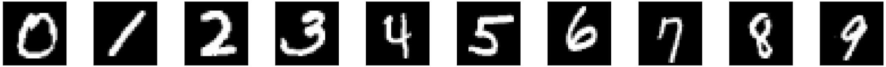
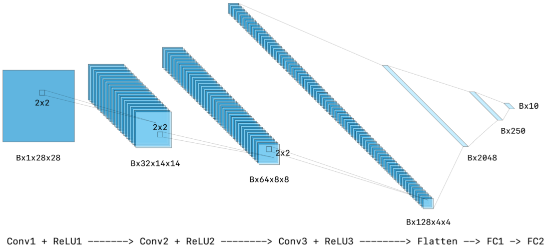


Figure: MNIST images

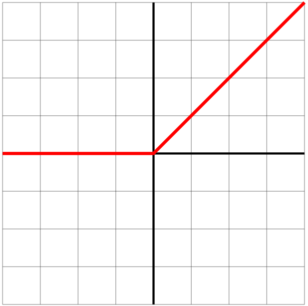
The Model

Convolutional Neural Network Architecture

1. **Convolutional + Bias + ReLU**: I: (B, 1, 28, 28), S:2, P:0, O: (B, 32, 14, 14)
2. **Convolutional + Bias + ReLU**: I: (B, 32, 14, 14), S:2, P:1, O: (B, 64, 8, 8)
3. **Convolutional + Bias + ReLU**: I: (B, 64, 8, 8), S:2, P:0, O: (B, 128, 4, 4)
4. **Fully Connected + ReLU**: I: (B,2048), O:(B,250)
5. **Fully Connected + SoftMax**: I: (B,250), O:(B,10)
6. **Loss**: Categorical Cross-Entropy



ReLU activation



$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

im2col approach

sliding_window_view

- Creates a new in-memory tensor with views of the given matrix as elements
- shape of views is specified as rows and columns

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{.sliding\_window\_view(:,(2,2))} \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix} \\ \begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix} & \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} \end{bmatrix}$$

reshape

- Change the shape of the array, matrix, tensor given as input in a **row-major** fashion:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \xrightarrow{.reshape(8,2)} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \end{bmatrix}$$

im2col convolution code

```python
def im2col_convolution(batch_of_images, kernels, stride, padding):
    # Pad the batch of images and extract shape of images and shape of kernels
    batch_of_images = np.pad(batch_of_images,((0,0),(0,0),(padding,padding),(padding,padding)))
    batch_size, input_channels, image_height, image_width = batch_of_images.shape
    kernels_number, kernel_channels, kernel_height, kernel_width = kernels.shape

    # Extract sliding windows from the input image, considering the kernel size and stride
    sliding_windows = np.lib.stride_tricks.sliding_window_view(batch_of_images,(1,input_channels,kernel_height,kernel_width))[:,:,::stride,::stride]

    # Reshape the windows to match the unrolled kernel's dimensions
    sliding_windows = sliding_windows.reshape((-1,(kernel_height * kernel_width * input_channels)))

    # Unroll the kernels
    kernels = kernels.reshape((-1,(kernel_height*kernel_width*input_channels))).transpose(1,0)

    # Dot product between images and kernels
    images_dot_kernels = np.matmul(sliding_windows, kernels).astype(np.float32)

    # Compute the output dimensions to reshape the resulting matrix (each row corresponds to a patch)
    output_width = int(((image_width - kernel_width) / stride) + 1)
    output_height = int(((image_height - kernel_height) / stride) + 1)

    # First operate a reshape keeping spatial ordering, which has channels at the end
    output = images_dot_kernels.reshape(batch_size, output_width, output_height, kernels_number)

    # Transpose to have input in shapes (batch, output_channel, height, width)
    output = output.transpose(0,3,1,2).astype(np.float32)
    return output
```

im2col optimized approach

Improvements

- **Less copies**: replacing .transpose() with .T and minimize th use of .reshape() to stop the creation of copies in memory, leads to less computation time.
- **BLAS-oriented memory management**: ensuring C-continuity for the first operand and F-continuity for the second one optimizes the computation.

Exclusive for the "im2col optimized BLAS" approach:

- **Direct calls to BLAS methods**: np.matmul() call BLAS methods, therefore calling them directly should reduce the overhead, specially for large matrices.

Exclusive for the "im2col CuPy" approach:

- **GPU exploitation**: All operations are done on the GPU to exploit fast matrix multiplications.

Performance evaluation of inference times

Premises

- Times are computed over 960 batches of 1 image or 30 batches of 32 images because of the high computation time of Nested Loops approach.
- **Im2Col CuPy** and **PyTorch Model** are **always** executed on the GPU, while the other approaches are **always** executed on the CPU.

Results for batch size = 1

Metric	Nested Loops	Im2Col (Unoptimized)	Im2Col Optimized	Im2Col Optimized BLAS	Im2Col CuPy	PyTorch Model
mean	0.905063	0.000473	0.000476	0.002521	0.004848	0.000705
std	0.158358	0.001717	0.001705	0.004707	0.005290	0.002592
min	0.846334	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.878515	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.885522	0.000000	0.000000	0.000000	0.003999	0.000000
75%	0.899730	0.000000	0.000000	0.002527	0.006017	0.000981
max	2.841252	0.016235	0.016589	0.031685	0.024350	0.034955

Table: Time results for 960 images

- im2col Unoptimized is **1913** times faster, on average, than nested loops approach.
- im2col Unoptimized performs better than PyTorch, with a decrease of **33%** in computation time.
- Optimizations are not exploited: optimized versions and GPU versions are slower.

Results for batch size = 32

Metric	Nested Loops	Im2Col (Unoptimized)	Im2Col Optimized	Im2Col Optimized BLAS	Im2Col CuPy	PyTorch Model
mean	28.797833	0.004053	0.005856	0.011154	0.005954	0.010989
std	0.262673	0.005367	0.005904	0.008674	0.007142	0.057249
min	28.443987	0.000000	0.000000	0.000000	0.000000	0.000000
25%	28.619262	0.000000	0.000000	0.003129	0.000145	0.000000
50%	28.750885	0.002197	0.004869	0.012423	0.003000	0.000000
75%	28.930829	0.005922	0.006992	0.016010	0.007788	0.001001
max	29.599773	0.016489	0.016596	0.028891	0.023160	0.314072

Table: Time results for 30 Batches

- im2col is **7105** times faster on average than nested loops approach.
- im2col Unoptimized performs better than PyTorch, with a decrease of **63%** in computation time.
- Optimizations are not exploited: optimized versions and GPU versions are slower.

Backward phase

ReLU derivative

Before being used to compute the gradients of the loss w.r.t. input, filter and bias, the gradient of the loss w.r.t. output $\frac{\delta L}{\delta O}$ must be element-wise multiplicated with a mask

$$X = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 10 & 1 \\ 1 & 1 & 1 \end{bmatrix}, K = \begin{bmatrix} 1 & 1 \\ 1 & -5 \end{bmatrix}$$

$$X \circledast K = \begin{bmatrix} -47 & 7 \\ 7 & 7 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 7 \\ 7 & 7 \end{bmatrix} \xrightarrow{Mask} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\frac{\delta L}{\delta O} = \begin{bmatrix} 1.23 & 3.59 \\ 4.65 & 0.81 \end{bmatrix} \xrightarrow{ReLUderive} \begin{bmatrix} 1.23 & 3.59 \\ 4.65 & 0.81 \end{bmatrix} \odot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 3.59 \\ 4.65 & 0.81 \end{bmatrix} = \delta_Z$$

Gradient w.r.t. image

- The gradient of the loss w.r.t. the image is needed to backpropagate the error to the preceding layers.
- It is obtained as:

$$\frac{\partial L}{\partial X} = \text{FullConvolution}(\delta_Z, W_{rot180^\circ})$$

Where X is the original image, W_{rot180} is the filter rotated by 180° and δ_Z is the gradient of loss w.r.t. the output.

Full Convolution becomes a standard **Convolution** between the properly dilated and padded gradient of output and the rotated kernel.

Gradient w.r.t. image

$$\mathrm{X}:\begin{bmatrix}1&2&3&4\\5&6&7&8\\9&10&11&12\\13&14&15&16\end{bmatrix},\quad \mathrm{W}:\begin{bmatrix}1&2\\3&4\end{bmatrix},\quad \text{Stride:2},\quad \text{Padding:0}$$

$$\delta_Z:\begin{bmatrix}2&1\\3&5\end{bmatrix}\xrightarrow{dilation(stride-1)}\begin{bmatrix}2&0&1\\0&0&0\\3&0&5\end{bmatrix}\xrightarrow{pad(stride-1)}\begin{bmatrix}0&0&0&0&0\\0&2&0&1&0\\0&0&0&0&0\\0&3&0&5&0\\0&0&0&0&0\end{bmatrix},\quad \mathrm{W}:\begin{bmatrix}1&2\\3&4\end{bmatrix}\xrightarrow{rotate(180°)}\begin{bmatrix}4&3\\2&1\end{bmatrix}$$

$$\frac{\partial L}{\partial X}=\begin{bmatrix}0&0&0&0&0\\0&2&0&1&0\\0&0&0&0&0\\0&3&0&5&0\\0&0&0&0&0\end{bmatrix}\circledast\begin{bmatrix}4&3\\2&1\end{bmatrix}$$

Gradient w.r.t. filter

- The gradient of the loss w.r.t. the filter is needed to update the filter's coefficients.
- It is the result of $Channels_{input} \times Channels_{output}$ **Single Channel Convolutions** between the input image and the gradient of the loss w.r.t. the output

Gradient w.r.t. filter

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 5 & 1 & 9 \\ 2 & 6 & 4 \\ 8 & 3 & 7 \end{bmatrix} \circledast \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 37 & 47 \\ 67 & 77 \end{bmatrix} \begin{bmatrix} 30 & 20 \\ 60 & 40 \end{bmatrix} \begin{bmatrix} 93 & 83 \\ 63 & 53 \end{bmatrix} \begin{bmatrix} 60 & 36 \\ 40 & 12 \end{bmatrix} \begin{bmatrix} 54 & 56 \\ 64 & 64 \end{bmatrix} \begin{bmatrix} 54 & 36 \\ 64 & 24 \end{bmatrix}$$

Gradient w.r.t. bias

- The gradient of the loss w.r.t. the bias is needed to update the bias's coefficients.
- It is computed as the sum of values of the loss w.r.t. the output for each channel

Performance evaluation of backpropagation times

Results for batch size = 1

Statistica	Nested Loops	Im2Col (Unoptimized)	Im2Col Optimized	Im2Col Optimized BLAS	Im2Col CuPy	PyTorch Model
mean	2.225652	0.006299	0.006389	0.009346	0.010289	0.001052
std	0.249752	0.005600	0.005081	0.008486	0.006828	0.003089
min	2.120881	0.000000	0.000000	0.000000	0.000000	0.000000
25%	2.168586	0.000000	0.002733	0.001243	0.005460	0.000000
50%	2.187272	0.006004	0.006017	0.008072	0.010371	0.000000
75%	2.220948	0.008441	0.008245	0.015309	0.015654	0.001000
max	5.059112	0.037697	0.031408	0.056891	0.034889	0.023702

Table: Time results for 960 images

- im2col is **353** times faster, on average, than nested loops approach
- im2col Unoptimized is slower than PyTorch, with an increase of **83%** in time.
- Optimizations are not exploited: optimized versions and GPU versions are slower.

Results for batch size = 32

Statistica	Nested Loops	Im2Col (Unoptimized)	Im2Col Optimized	Im2Col Optimized BLAS	Im2Col CuPy	PyTorch Model
mean	70.565303	0.043744	0.042249	0.060530	0.019055	0.006732
std	0.405636	0.007077	0.006777	0.007447	0.015966	0.028690
min	69.947701	0.027079	0.028126	0.044401	0.000000	0.000000
25%	70.319230	0.039957	0.038119	0.055597	0.006083	0.000000
50%	70.498324	0.045567	0.042143	0.063513	0.012918	0.001000
75%	70.760394	0.049187	0.047878	0.066385	0.031700	0.001672
max	71.698621	0.054494	0.056850	0.072984	0.052906	0.157841

Table: Time results for 30 Batches

- im2col Unoptimized is **1613** times faster on average than nested loops approach.
- Optimizations and GPU are finally exploited: PyTorch is leading, but optimizing im2col brings a reduction of **3%** and exploiting the GPU brings a total reduction of **56%** in computation time.

Conclusions

- im2col approach brings a significant boost in performance with respect to a nested loop approach.
- Optimizing the code provide time saving when large tensors are involved.
- GPUs increase performances when they are sufficiently utilized.
- There is space for further testing with bigger inputs and bigger networks.

THANK YOU !